EWAS 2006

3rd European Workshop on Aspects in Software

University of Twente, Enschede, the Netherlands
August 31, 2006

Günter Kniesel (Editor)

Technical Report IAI-TR-2006-6
ISSN 0944-8535

Institut für Informatik III
Universität Bonn

Made in MAKAO

Bram Adams

Bram.Adams@UGent.be
http://users.ugent.be/~badams/makao/
Ghislain Hoffman Software Engineering Lab
INTEC, Ghent University, Belgium

ABSTRACT

Software systems do not solely consist of source code: various other types of artifacts play a role, notably the build system. Although nothing stands as close to the source code or lends itself better as a starting point to explore a system's high-level architecture, few techniques or tools recognize or exploit this. Still, as each considerable change to a software system potentially demands modifications of build-related files, maintainers and developers alike need to find their way around quickly. We present MAKAO (Makefile Architecture Kernel for Aspect Orientation), a re(verse)-engineering framework for build systems, as a means to extract as much knowledge from the build system as possible and to help solving typical build-related problems. MAKAO offers a DAG (Directed Acyclic Graph)-based view of a build system, supporting visualization, querying and (dynamic) modification.

1. BUILD SYSTEMS

Until 1977, ad hoc build and install scripts were used to automate the build process of software systems. Everything changed when Feldman presented "make" [1], the most influential software build tool ever. He proposed a declarative way of specifying the dependencies between targets (executables, object files, libraries, source files, etc.), whereas the recipe to build a target was written as an imperative list of commands and macros. The "make" interpreter relied on the observation that a target only needs to be (re)built by its recipe if at least one of its dependencies is newer. This greatly improved incremental compilation of software projects.

Later on, portability of software required configurability of both source code and build scripts. Figure 1 illustrates this. Configuration scripts written in e.g. GBS (GNU Build System)[1] describe the build process from a high-level perspective, abstracting away from platform-specific configuration issues. Afterwards, they generate the actual build scripts that will perform the ground work. The combination of both is called the build system.

Build systems play a crucial role, as various stakeholders interact with it, each with their own interests and problems:

developers Assess the effects of their code or, if the build did not succeed, try to find out what error was the culprit. When adding new source code, they want to find out where they need to change something.

maintainers Want to learn the inner mechanics of a new system, check if there is dead code, profile things, etc.

(power) users Try to find out what library dependencies they need to compile and run the software.

QA division Want to add feature and regression tests and run them as quickly as possible.

[1] http://sources.redhat.com/autobook/

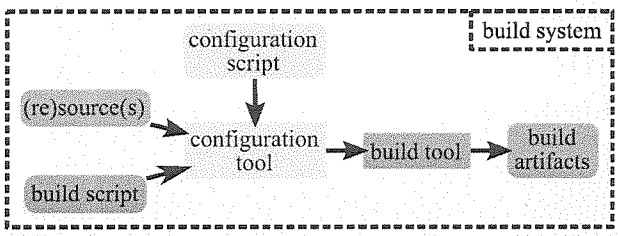

Figure 1: High-level view of build systems.

researchers Try to uninvasively integrate experimental tools.

As a consequence, the build process implicitly contains valuable information about all facets of the software itself. We would like to extract this inherent build system knowledge and make it available in an explicit form to all stakeholders. This way, they can deal more succinctly with their problems and learn new things about the software system's architecture. This enhances the data gained by existing reverse-engineering techniques for source code.

In the remainder of this abstract, we will present MAKAO (section 2) and apply it on the case study of [3] (section 3).

2. MAKAO

The observations of the previous section made us think about an extensible visualisation and re(verse)-engineering framework for build systems, which we named the Makefile Architecture Kernel for Aspect Orientation (MAKAO).

2.1 Design

MAKAO's philosophy is to make the implicit explicit. The declarative nature of dependency specifications is a good thing, but these are typically spread over hundreds of files and composed in some unclear way (e.g. recursively [2]). A visual representation combined with a powerful querying facility would be able to clarify this. However, more invasive problems like the addition of new tools (see section 3) also require re-engineering of the build system. This involves e.g. introducing new build targets, adding extra dependencies to existing targets or modifying targets' recipes. The design of MAKAO takes both the reverse as well as the re-engineering functionality into account.

Inspired by the ideas of aspect-oriented programming (AOP), MAKAO is composed of the following four components:

Explorer (Visually) Explore a representation of the build system.

Finder Query for targets and commands based on properties.

Adviser Write modifications for targets' dependencies and recipes.

Weaver Apply modifications both logically (in-memory representation) and physically (build and configuration scripts).

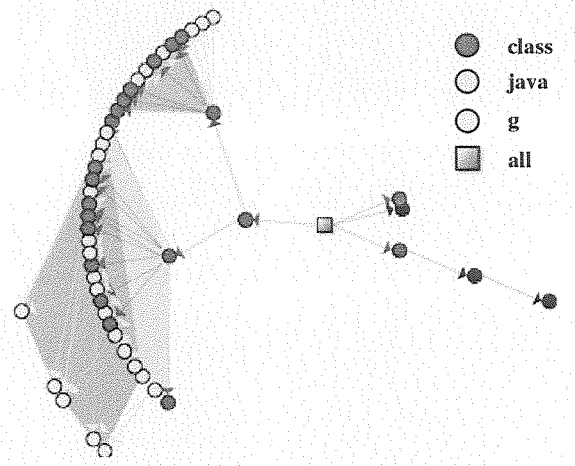

Figure 2: Dependency graph of build of Aspicere.

2.2 Data model

We opted for a Directed Acyclic Graph (DAG) as the underlying data model of a build run, based on the following observations:

- DAGs form the underlying model of "make" [1], and hence of most of its successors.
- Graphs have a natural visual representation and can be easily modified.
- Configuration scripts define in fact templates for source code and build scripts.
- Stakeholders, however, are mostly confronted with instantiated, platform-specific build scripts and code.

Providing only static views of a build system would not be useful, as the templates are too general and hence too hard to navigate or understand. Instead we chose, influenced by the fourth observation, to represent dynamic traces of concrete builds, but with links back to the static build data. Basically, while performing a typical build the build tool's internally constructed dependency graph is extracted and fed into MAKAO. In practice, this can be obtained by using a modified "make" like "remake"[2] or e.g. by capturing and post-processing the debug output produced by the build tool.

2.3 Implementation

Fig. 2 shows the dependency graph of a full build of Aspicere [3], as seen in MAKAO's main panel. We implemented MAKAO on top of GUESS[3], a graph exploration tool with an embedded Jython-based scripting language (called Gython). Targets (nodes) have directed edges to their dependencies. Each build script is shown as a colored convex hull around all of its targets, unless the hull degenerates to e.g. one node.

An additional "legend" panel, pasted on the graph here for the reader's convenience, outlines the various known build concerns in use (".java", ".o", etc.) and assigns a color to them. Unknown concerns are colored black by default, while the dark blue targets really represent the same concern as the square (starting) node's one. Nodes, edges and hulls are simply objects and can be easily queried and manipulated in the scripting console (not shown).

3. EXAMPLE

We will now show MAKAO's use by applying it on an issue we encountered during a reverse-engineering experiment [3] using Aspicere, our aspect language for C. To weave a tracing aspect into the code base, we had to run Aspicere's source preprocessing weaver right before each file's compilation. Doing this by replacing the relevant commands by a wrapper script around the aspect weaver did not resolve subtle issues like two or more wrappers invoking each other. Manually modifying the makefiles seemed inevitable back then, but we will now revisit this case with MAKAO.

First, we build the unaltered system while we extract the constructed dependency graph. After loading this graph into MAKAO, the Explorer-component allows us to verify that there are indeed C source files and that they apparently reside in a small number of directories. We are interested in those targets T directly depending on (i.e. processing) C files and we want to insert calls to Aspicere's weaver in T's recipe before the (sole) source-processing command. We write the following queries in the Finder-component:

```
1 T_list=[e.getNode1() for n in (concern=="c")
                       for e in n.getInEdges()    ]
3 base=[(command,tool) for T in T_list
                       for command in commands[T]
5                      for tool in ["gcc","esql"]
                       if command.find(tool)!=-1 ]
```

On line 1, we select all source nodes of edges leading to C targets, presuming that there are no duplicates. Then we try to find each T's sole source-processing command in its recipe using a Gython list comprehension. The tools we mention in this query could have been discovered earlier on by querying too. Now, we can write advice using the Adviser:

```
  before_advice=
7     ["\n".join([c.replace(t,t+"␣-E␣-o␣${<}"),
                  "aspicere.sh␣${<}"])
9           for (c,t) in base                    ]
```

The `join` function concatenates invocations of the selected tool in preprocessing mode and of Aspicere's weaver. Finally, the Weaver should weave all these concatenated commands in the recipes of the T targets as before advice, both in MAKAO's memory representation as in the proper build and/or configuration scripts:

```
9 weave_before(T_list,[c for (c,t) in base],
                 before_advice)
```

One can now run the modified build scripts, check the scripts themselves or issue some new queries on MAKAO's memory model to see whether any relevant targets were skipped.

4. CONCLUSION

Build systems are inherently part of and tied to software systems. They offer valuable architectural information to various stakeholders. To facilitate quick understanding and clever modifications, MAKAO offers visualisation and flexible manipulation of both build structure and behavior inspired by aspect-oriented techniques.

Acknowledgements

The author wants to thank Kris De Schutter, Andy Zaidman and Herman Tromp for their support and (cumulative) wisdom.

5. BIBLIOGRAPHY

[1] S. I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 1979.

[2] P. Miller. Recursive make considered harmful, 1997.

[3] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006.

[2] `http://bashdb.sourceforge.net/remake/`

[3] `http://graphexploration.cond.org/`